Stratégies de dérivations de programmes logiques

Francis Alexandre, Khaled Bsaïes

Stratégies de dérivations de programmes logiques

Francis Alexandre[*] et Khaled Bsaïes[**]

[*] LORIA - Université Henri Poincaré
BP 239, 54506 Vandœuvre-lès-Nancy, France
e-mail : alexandr@loria.fr

[**]Faculté des sciences de Tunis Université de Tunis II
1060 Le Belvédère Tunis, Tunisie
e-mail : Khaled.Bsaies@fst.rnu.tn

Résumé

Cet article traite de la construction de programmes logiques (programmes en clauses de Horn) et plus particuliérement du problème des eurêkas. Dans le processus de transformation de programmes, il est souvent nécessaire d'utiliser certaines propriétés des programmes. Ces propriétés sont appelées eurêkas, car elles ne sont pas données a priori, ce sont des propriétés intrinsèques aux programmes et c'est l'utilisateur qui doit les découvrir. Le problème est donc crucial, car d'une part sans ces eurêkas il n'est pas possible de poursuivre la transformation, d'autre part l'invention de tels eurêkas, qui repose sur l'intuition et l'expertise de l'utilisateur, est difficile à automatiser.

La technique présentée ici est une tentative d'automatisation de la génération de ces eurêkas. Cette technique doit être considérée comme une contribution à l'automatisation dans la domaine de la construction de programmes par transformation, elle vient donc compléter les nombreuses tactiques existant déjà dans ce domaine.

1 Introduction

Cet étude s'inscrit dans le domaine de la construction de programmes logiques par transformation.

La démarche est la suivante: l'informaticien écrit "à la main" une première spécification de son problème en utilisant le langage des clauses de Horn. Il vérifie que cette spécification possède certaines propriétés, par exemple qu'elle est bien typée, il peut aussi s'assurer que sa spécification est correcte par rapport au problème donné en effectuant des tests.

En général cette spécification initiale n'est pas efficace car des prédicats et des structures "intermédiaires" ont été utilisés. C'est souvent le cas, lorsque l'on spécifie en utilisant une approche descendante, c'est à dire lorsque pour résoudre le problème initial on a été conduit à considérer plusieurs sous-problèmes.

C'est pour cette raison qu'il est nécessaire de considérer une seconde étape dans laquelle on transformera la spécification initiale en un programme plus efficace. L'approche transformationnelle [12] consiste à transformer pas à pas un énoncé en appliquant à chaque étape des transformations, jusqu'à obtenir un énoncé final que l'on juge satisfaisant. Le système de transformation que nous utilisons est celui décrit dans [17], il est basé sur les transformations de dépliage et de pliage. Comme la correction de ces transformations a été démontrée, la problèmatique consiste essentiellement à trouver des stratégies d'application des transformations. Un point crucial est le problème des "eurekas", ces "eurekas" sont soit de nouveaux prédicats à définir, soit des propriétés des programmes, c'est ce dernier cas que nous traitons dans cet article.

Illustrons par un exemple la problématique de la découverte de ces propriétés.

Soit DIV l'ensemble des clauses de Horn suivantes, spécifiant la division euclidienne:

(1) $div(a,b,q,r) \leftarrow fois(q,b,t),\ plus(t,r,a),\ inf(r,b)$

(2) $plus(0,x,x) \leftarrow$
(3) $plus(s(x),y,s(z)) \leftarrow plus(x,y,z)$

(4) $fois(0,x,0) \leftarrow$
(5) $fois(s(x),y,z) \leftarrow fois(x,y,u),\ plus(u,y,z)$

(6) $inf(0,s(x)) \leftarrow$
(7) $inf(s(x),s(y)) \leftarrow inf(x,y)$

Les symboles fonctionnels 0 et s sont les constructeurs des entiers naturels. Les symboles de prédicats div, $plus$, mul et inf définissent respectivement la division euclidienne, l'addition, la multiplication et la relation $<$ dans les entiers naturels. La clause (1) exprime que q et r sont respectivement le quotient et le reste de la division de a par b. Cette spécification, même si elle est exécutable, est très inefficace ceci est dû essentiellement au caractère déclaratif de cette spécification. Un des buts de la transformation est donc de supprimer certains prédicats intermédiaires que l'on a utilisés pour décrire le problème, mais qui rendent l'exécution du programme inefficace. Pour atteindre ce but il est en général nécessaire d'obtenir une définition récursive des prédicats. Dans ce cas une définition récursive de div ne faisant plus intervenir le prédicat $fois$ permet d'obtenir une définition plus efficace. L'obtention d'une définition récursive se fait par application des transformations de dépliage et pliage.

Montrons sur cet exemple comment fonctionnent les transformations de dépliage et pliage. La transformation de dépliage consiste à évaluer un atome dans le corps d'une clause en utilisant la règle de résolution. Ainsi l'atome $fois(q,b,t)$ de la clause (1) peut s'unifier avec les atomes $fois(0,x,0)$ et $fois(s(x),y,z)$, têtes des clause (4) et (5), en appliquant la règle de

résolution entre les clauses (1) et (4) d'une part, puis les règles (1) et (5) d'autre part, on obtient les clauses suivantes :

$$\begin{array}{lll} (8) & div(a,b,0,r) & \leftarrow \quad plus(0,r,a),\ inf(r,b) \\ (9) & div(a1,y,s(x),r1) & \leftarrow \quad fois(x,y,u),\ plus(u,y,z), \\ & & \qquad plus(z,r1,a1),\ inf(r1,y) \end{array}$$

Dans le processus de transformation la clause (1) est remplacée par les clauses (8) et (9), ce remplacement préserve la sémantique des programmes (au sens du plus petit modèle de Herbrand). Dans la clause (8) on remarque que l'atome $plus(0,r,a)$ contient le terme clos 0, lorsque l'on est dans ce cas il est souvent intéressant de continuer les transformations de dépliages, ici le dépliage de l'atome $plus(0,r,a)$ (en utilisant la clause (2)) produit la clause (10) suivante, qui remplacera la clause (8) dans le processus de transformation :

$$(10)\ div(a,b,0,a) \leftarrow \quad inf(a,b)$$

La suite de la transformation se focalise sur la clause (9), le but est alors d'obtenir une définition récursive du prédicat div. C'est là qu'intervient alors la transformation de pliage. La transformation de pliage consiste à remplacer dans le corps d'une clause (corps $\equiv$ partie droite), l'instance du corps d'une clause par l'instance correspondante de sa tête. Concrétement dans le corps de la clause (9), il faut retrouver une instance du corps de la clause (1). Comparons Γ et Λ les corps de (1) et (9).

$$\begin{array}{lllll} \Gamma\ : & fois(q,b,t), & plus(t,r,a), & & inf(r,b) \\ \Lambda\ : & fois(x,y,u), & plus(u,y,z),\ plus(z,r1,a1), & & inf(r1,y) \end{array}$$

On peut remarquer que :

- $fois(x,y,u)$, $plus(u,y,z)$, $inf(r1,y)$ n'est pas une instance de Γ (à cause de la variable r de Γ, qui ne peut être substituée simultanément par les variables y et $r1$).
- $fois(x,y,u)$, $plus(z,r1,a1)$, $inf(r1,y)$ n'est pas une instance de Γ (à cause de la variable t de Γ, qui ne peut être substituée simultanément par les variables u et z).

Pour ces deux raisons il est donc impossible d'effectuer la transformation de pliage. Dans cet exemple on est donc amené à modifier le corps de la clause (9) en utilisant les propriétés des prédicats. Les prédicats $fois$ et $plus$ définissent respectivement la multiplication et l'addition dans les entiers naturels et l'on sait que ces opérations ont en particulier les propriétés de commutativité et d'associativité. On utilise ces deux propriétés pour le prédicat $plus$:

$plus(u,y,z)$, $plus(z,r1,a1)$ signifie que $(u + y) + r1 = a1$ (z étant une variable "intermédiaire"). En utilisant le fait que l'addition est une opération commutative et associative, on peut écrire que $(u + r1) + y = a1$, ce qui se traduit sous forme de prédicats par $plus(u,r1,w)$, $plus(w,y,a1)$, si l'on effectue le remplacement de $plus(u,y,z)$, $plus(z,r1,a1)$ par $plus(u,r1,w)$, $plus(w,y,a1)$, dans le corps de (9) on obtient

$$\begin{array}{lll} (11) & div(a1,y,s(x),r1) & \leftarrow \quad fois(x,y,u),\ plus(u,r1,w), \\ & & \qquad plus(w,y,a1)\ inf(r1,y) \end{array}$$

Le pliage de (1) dans (11) devient alors possible et produit la clause suivante :

$$(12)\quad div(a1,y,s(x),r1) \leftarrow \quad plus(w,y,a1),\ div(w,y,x,r1)$$

La définition obtenue, pour la division euclidienne est la suivante :

(10) $div(a,b,0,a) \leftarrow inf(a,b)$
(12) $div(a1,y,s(x),r1) \leftarrow plus(w,y,a1),\ div(w,y,x,r1)$

Cette définition est évidemment beaucoup plus efficace que la spécification initiale. L'exécution d'un but de la forme $div(a,b,q,r)$ où les a et b sont des termes clos et q, r des variables va consister à "soustraire" b de a jusqu'à ce que a soit strictement inférieur à b.

Cette description nous conduit aux affirmations suivantes :

- L'utilisation des propriétés des programmes est parfois indispensable à la transformation des programmes. Pour cet exemple, à notre connaissance, il n'existe pas de technique transformationnelle permettant d'obtenir un programme aussi efficace.
- Un problème important est la découverte des propriétés. Dans l'exemple proposé on utilise les propriétés classiques des opérations arithmétiques, on pourrait adjoindre ces propriétés à la spécification initiale. Il existe des cas dans lesquels les propriétés sont moins classiques, il est alors indispensable de les découvrir lors du processus de transformation.

La suite de l'article est structurée de la façon suivante : le paragraphe 2 donne les notations et les définitions de base ainsi que la description du démonstrateur utilisé, le paragraphe 3 est une description de la technique utilisée, le paragraphe 4 montre sur deux exemples comment fonctionne la technique, enfin le paragraphe 5 conclut par une discussion et les perspectives possibles de ce travail.

2 Description du démonstrateur

2.1 Définitions de base

Dans ce paragraphe nous définissons quelques notations concernant les notions de base de la logique [11]. Si e est une expression nous notons $\mathcal{V}(e)$ l'ensemble des variables ayant des occurrences dans e. Dans la suite nous notons les atomes A, B, $C, \ldots$, les conjonctions d'atomes Λ, Δ, Γ, $\Phi, \ldots$ et les substitutions θ, $\sigma, \ldots$. Une substitution est aussi notée sous la forme $\{x_1/t_1, \ldots, x_n/t_n\}$, l'ensemble des variables $\{x_1, \ldots, x_n\}$ est appelé le domaine de la substitution. pgu est une abréviation pour *plus général unificateur*.

Un programme ou (programme défini) P est un ensemble de clauses de Horn, $\mathcal{M}(P)$ désigne le plus petit modèle de Herbrand de P. Nous considérons des formules particulières, appelées formules implicatives. Ces formules sont particuliérement importantes pour l'approche décrite, elle permettent de modéliser les propriétés des prédicats.

Définition 2.1 (Formule implicative) *Une formule implicative est une formule du premier ordre de la forme:*

(1) $\forall x_1,\ldots,\forall x_n(\exists y_1,\ldots,\exists y_m\ \Delta \leftarrow \exists z_1,\ldots,\exists z_k\ \Lambda)$ *où:*
Δ *et* Λ *sont des conjonctions d'atomes.*
$\{x_1,\ldots,x_n\} = \mathcal{V}(\Delta) \cap \mathcal{V}(\Lambda)$,
$\{y_1,\ldots,y_m\} = \mathcal{V}(\Delta) \setminus \mathcal{V}(\Lambda)$ *and*
$\{z_1,\ldots,z_k\} = \mathcal{V}(\Lambda) \setminus \mathcal{V}(\Delta)$

Une telle formule est notée $\Delta \leftarrow \Lambda$ *s'il n'y a pas d'ambiguité.*

Notons que les clauses de Horn sont des cas particuliers de formules implicatives.

Les variables quantifiées existentiellement sont aussi appelées les variables internes, les autres sont quantifiées universellement.

Définition 2.2 (Propriété d'un programme) *Soient P un programme et $\pi : \Delta \leftarrow \Lambda$ une formule implicative, π est une propriété de P si π est valide dans $\mathcal{M}(P)$.*

Les remplacements d'atomes dans les corps des clauses peuvent être vus comme des applications de propriétés des programmes. Par exemple si l'on considére la propriété $plus(x,y,t), plus(t,z,r) \leftarrow plus(y,z,u), plus(x,u,r)$, qui exprime en partie l'associativité du prédicat $plus$, il est possible de remplacer une instance de la partie gauche de la formule par l'instance correspondante de la partie droite.

2.2 Description du démonstrateur

Le démonstrateur que nous présentons est une adaptation de celui décrit dans [16]. Il permet de prouver qu'une formule implicative est valide dans le plus petit modèle de Herbrand d'un programme défini.

Nous considérons cinq règles de déduction. Une règle de déduction génére un ensemble de formules implicatives à partir d'autres formules implicatives et des clauses de la spécification. Ce démonstrateur peut aussi être vu comme une extension du système de transformation de Tamaki et Sato [17]. Dans la suite lorsque nous parlerons de formules, il s'agira de formules implicatives.

Soit P un programme défini et $\pi : \Delta \leftarrow \Gamma$ une formule. Pour prouver que π est valide dans $\mathcal{M}(P)$, l'idée est d'appliquer des règles à chaque membre de la formule, dans le but d'obtenir des formules de la forme $\Gamma \leftarrow \Gamma,\Delta$ ou $\Gamma \leftarrow false$ (c'est à dire la formule vraie $true$).

Nous décrivons les règles de déduction sous la forme de règles d'inférence $\frac{\pi}{S''}(NOM)$, où π est une formule et S'' est un ensemble de formules, NOM est le nom de la règle.

Le processus de déduction est le suivant. A chaque étape, nous avons un ensemble S de formules à prouver. Nous choisissons une formule π de S et nous appliquons une règle de déduction à π, ensuite nous remplaçons la formule choisie par l'ensemble de formules S'' généré par la règle de déduction. Nous obtenons un nouvel l'ensemble S' de formule à prouver : $S' = (S \setminus \{\pi\}) \cup S''$.

Pour chaque règle, S' est donné en fonction de S.

Définition 2.3 (Simplification)

$$\frac{\pi \ : \ \Delta, A \ \leftarrow \ \Gamma, B}{(\Delta \ \leftarrow \ \Gamma)\theta} \ (SIMP)$$

θ est une substitution $\{x_i/t_i\}$ telle que $A\theta = B$, le domaine de θ est constitué de l'ensemble des variables internes de A et des variables de t_i ayant des occurrences dans Γ, B. Nous avons $S' = (S \setminus \{\pi\}) \cup \{(\Delta \ \leftarrow \ \Gamma)\theta\}$.

La simplification permet de supprimer des atomes des deux côtés de la formule. Cette règle peut être vue comme une heuristique, parce qu'elle ne préserve pas l'équivalence, S peut être valide dans $\mathcal{M}(P)$ tandis que S' ne l'est pas, bien sur si S' est valide alors S l'est aussi.

Définition 2.4 (Dépliage droit)

$$\frac{\pi \ : \ \Gamma \ \leftarrow \ \Delta,A}{\{ \ \pi_i \ : \ (\Gamma \ \leftarrow \ \Delta,\Delta_i)\theta_i, \ i \in [1,\ldots,k]\}} \ (UNF\_R)$$

$E = \{c_1,\ldots,c_k\}$ est l'ensemble des clauses du programme P telles que $c_i : \ B_i \leftarrow \Delta_i$ et il existe $\theta_i = pgu(B_i,A)$.

Si $E = \emptyset$, alors nous générons la formule $\Gamma \leftarrow false$, que nous pouvons réduire à la formule true. Nous avons $S' = (S \setminus \{\pi\}) \cup \{ \pi_i , i \in [1,\ldots,k]\}$.

Notons que pour prouver π il est nécessaire de prouver toutes les formules π_i.

Définition 2.5 (Dépliage gauche)

$$\frac{\pi \ : \ \Gamma,A \ \leftarrow \ \Delta}{\{\pi_i \ : \ (\Gamma, \Delta_i)\theta_i \leftarrow \ \Delta, \ i \in [1,\ldots,k]\}} \ (UNF\_L)$$

$E = \{c_1,\ldots,c_k\}$ est l'ensemble des clauses du programme P telles que $c_i : \ B_i \leftarrow \Delta_i$ et telles qu'il existe $\theta_i = pgu(B_i,A)$, où θ_i sont des pgu existentiels (c'est à dire dont les domaines contiennent seulement des variables de A et des variables de B). Nous avons $S' = (S \setminus \{\pi\}) \cup \{ \pi_{i_0}\}$ pour un $i_0 \in [1,\ldots,k]$.

Pour prouver la formule initiale π, il est suffisant de prouver une des formules π_i générée par le dépliage gauche.

Définition 2.6 (Coupure droite)

$$\frac{\begin{array}{c}\Lambda \leftarrow \ \Pi \\ \pi \ : \ \Gamma \ \leftarrow \ \Delta_1,\Delta_2\end{array}}{\pi' : \ \Gamma \leftarrow \Lambda\theta,\Delta_2} \ (CUT\_R)$$

- *θ est une substitution telle que $\Pi\theta = \Delta_1$*
- *pour toute variable x de Π, $x\theta$ est une variable qui n'a pas d'occurrences ailleurs que dans $\Pi\theta$*
- *θ substitue les différentes variables internes de Π par différentes variables internes de Δ_1,Δ_2.*
- *la formule $\Lambda \leftarrow \ \Pi$ apparaît dans un précédent ensemble de formules, mais pas dans l'ensemble courant de formule.*

Nous avons $S = (S' \setminus \{\pi\}) \cup \{\pi'\}$.

La coupure droite est une généralisation de la transformation classique de pliage définie dans [17].

Définition 2.7 (Coupure gauche)

$$\frac{\begin{array}{c}\Lambda \leftarrow \ \Pi \\ \pi \ : \ \Gamma_1, \Gamma_2 \ \leftarrow \ \Delta\end{array}}{\pi' \ : \ \Pi\theta,\Gamma_2 \leftarrow \Delta} \ (CUT\_L)$$

- *θ est une substitution telle que $\Lambda\theta = \Gamma_1$*
- *pour toute variable x de Λ, $x\theta$ est une variable et n'a pas d'occurrence ailleurs que dans $\Lambda\theta$.*
- *θ substitue les différentes variables internes de Λ par différentes variables internes de Γ_1,Γ_2*
- *la formule $\Lambda \leftarrow \ \Pi$ apparaît dans un précédent ensemble de formules mais pas dans l'ensemble courant de formules.*

Nous avons $S = (S' \setminus \{\pi\}) \cup \{\pi'\}$.

Notons que les règles de coupures (gauche et droit) sont seulement des heuristiques, elles permettent de faire des preuves par récurrence. Les formules $\Lambda \leftarrow \Pi$ sont les hypothèses de récurrence. Ainsi il est généralement nécessaire d'appliquer les règles de dépliages avant d'utiliser les règles de coupures, dans le but de faire décroître certains arguments par rapport à un ordre approprié.

3 Description générale de la technique

Soit P un programme donné et $c : p(\overline{t}) \leftarrow \Gamma$ une clause définie telle que le symbole p n'a ni d'occurrence dans P, ni dans Γ et où $\overline{t} = t_1, \ldots, t_n$.

La technique proposée a pour but de synthétiser une définition récursive de p, elle consiste en deux parties. La première partie est la génération de schémas, la seconde est l'instanciation de ces schémas en utilisant le démonstrateur décrit précédemment.

3.1 Génération de schémas

Etant donnée la clause ininitiale $c : p(\overline{t}) \leftarrow \Gamma$, la première phase consiste à effectuer des dépliages dans les atomes de c, on obtient ainsi une clause $c' : p(\overline{t}\theta) \leftarrow \Gamma$ où θ est la substitution obtenue par composition de toutes les substitutions correspondant aux différents pliages. Ceci est résumé comme suit :

$$
\begin{array}{ll}
c & : \; p(\overline{t}) \leftarrow \Gamma \\
 & \quad \ldots \\
\downarrow & \textit{dépliages} \\
 & \quad \ldots \\
c' & : \; p(\overline{t}\theta) \leftarrow \Lambda
\end{array}
$$

Dans cette phase il existe de nombreuses façons d'effectuer les dépliages. Le problème des dépliges a été étudié dans [13, 14], dans une autre problématique appelée "loop absorbtion" en vue de définir de nouveau prédicats.

En ce qui concerne cette stratégie, la manière d'effectuer les dépliages est guidée par les têtes de clauses obtenues. En effet, les transformations de dépliages introduisent en général des constructeurs dans les têtes de clauses. Dans l'exemple de l'introduction, le dépliage de l'atome $fois(q,b,t)$ de la clause (1) produit la clause :

$$
(9) \quad div(a1,y,s(x),r1) \leftarrow fois(x,y,u),\ plus(u,y,z),\ plus(z,r1,a1),\ inf(r1,y)
$$

Nous remarquons que le troisième argument de $div(a1,y,s(x),r1)$ est alors $s(x)$, cela correspond à chercher une définition par récurrence en faisant une induction sur le quotient.

Un dépliage des atomes $plus(t,r,a)$ ou $inf(r,b)$ aurait produit les têtes de clauses $div(s(a),b,q,r)$ et $div(a,s(b),q,s(r))$, ce qui correspond à des inductions sur le dividende et le diviseur. Dans cette étape c'est donc l'utilisateur, qui choisit la définition, qu'il désire obtenir.

Les dépliages ayant été effectués, en général il n'est pas possible de faire un pliage de c dans c', car dans Λ on ne trouve d'instance de Γ. Par contre, si l'on se limite à une partie Γ' des atomes de Γ, il est possible de trouver dans Λ une instance, Λ', de Γ' . On a donc la situation suivante :

$\Gamma = \Gamma', \Gamma''$
$\Lambda = \Lambda', \Lambda''$ *avec une substitution* σ *telle que* $\Lambda' = \Gamma'\sigma$

et les clauses

$c : p(t) \leftarrow \Gamma', \Gamma''$
$c' : p(t\theta) \leftarrow \Lambda', \Lambda''$

On dit que l'on effectue partiellement le pliage de c dans c' et l'on obtient le schéma suivant :

$p(t\theta) \leftarrow p(t\sigma), \Delta$ *où* Δ *est une conjonction d'atomes à déterminer.*

Remarques 3.1

La conjonction d'atomes Λ'' *n'est pas significative dans la clause obtenue par le pliage partiel, elle a donc été supprimée.*

Quelques critères permettent de déterminer Γ' *:*

- *le nombre d'atomes de* Γ' *: on choisit* Γ' *avec le maximum d'atomes.*
- *la substitution* σ *:* Γ' *peut être choisi de façon à ce que* σ *instancie un nombre maximum de variables de* Γ.

3.2 Instanciation du schéma

Etant donné le schéma π :

$$\pi : p(t\theta) \leftarrow p(t\sigma), \Delta$$

le problème est de trouver Δ tel que π soit valide dans le plus petit modèle de Herbrand de $P \cup \{c\}$.

La résolution de ce problème se fait en utilisant le démonstrateur. Remarquons que le problème de démontrer une formule implicative est évidemment indécidable, a fortiori il en est de même de l'invention de Δ dans le schéma.

Si le problème fait intervenir des types définis par récurrence, comme les entiers ou les listes, la formule π que l'on doit instancier est souvent une formule qui se démontre par récurrence. C'est pourquoi la stratégie générale que l'on applique consiste à considérer les cas terminaux, puis à étendre au cas général. L'utilisateur est guidé par les deux éléments suivants :

- trouver une formule de la forme $\Gamma \leftarrow \Gamma, \Delta$, qui est un théoréme.
- utiliser les conditions syntaxiques des règles d'application du démonstrateur et les conditions syntaxiques (en général des considérations sur les variables) du schéma à instancier.

Le paragraphe suivant illustre comment fonctionne cette stratégie.

4 Exemple

Dans ce paragraphe nous donnons deux exemples pour illustrer la technique, nous insistons sur l'instanciation des schémas. Nous ne développons pas les preuves des formules générées. Dans les exemples, nous soulignerons les atomes qui sont dépliés ou simplifiés.

4.1 Exemple de la division euclidienne

1. Génération du schéma suivant : $div(a,b,s(q),r) \leftarrow \Delta$, $div(u,b,q,r)$, et nous supposons que $\mathcal{V}(\Delta) \subset \{a,b,q,r,u\}$. (nous avons renommé le schéma pour plus de lisibilité).
2. Instanciation du schéma : $div(a,b,s(q),r) \leftarrow \Delta$, $\underline{div(u,b,q,r)}$

   $\Downarrow$ dépliage à droite

   $\underline{div(a,b,s(q),r)} \leftarrow \Delta$, $times(q,b,t)$, $plus(t,r,u)$, $less(r,b)$

$\Downarrow$ dépliage à gauche

$times(s(q),b,v),\ \ plus(v,r,a),\ \ less(r,b)$
$\leftarrow \Delta,\ times(q,b,t),\ \ plus(t,r,u),\ \ less(r,b)$

$\Downarrow$ simplification de l'atome $less(r,b)$

$\underline{times(s(q),b,v)},\ \ plus(v,r,a) \leftarrow \Delta,\ times(q,b,t),\ plus(t,r,u)$

$\Downarrow$ dépliage à gauche

$\underline{times(q,b,w)},\ plus(w,b,v),\ plus(v,r,a) \leftarrow \Delta,\ \underline{times(q,b,t)},\ plus(t,r,u)$

$\Downarrow$ simplification en utilisant la substitution existentielle $\{w/t\}$.

$plus(t,b,v),\ plus(v,r,a) \leftarrow \Delta,\ plus(t,r,u) \quad (C)$

Il est impossible d'appliquer la règle de simplification dans la formule C, car les variables t, b, r and a sont universellement quantifiées.
$\Downarrow$ dépliage droit (atome $plus(t,r,u)$)

$$\begin{cases} plus(0,b,v),\ plus(v,r,a) \leftarrow \Delta\theta_1 \\ plus(s(t),b,v),\ plus(v,r,a) \leftarrow \Delta\theta_2,\ plus(t,r,u) \end{cases}$$

where $\theta_1 = \{t/0,u/r\}$ and $\theta_2 = \{t/s(t),u/s(u)\}$.

En considérant le cas terminal, nous avons la formule :
$\underline{plus(0,b,v)},\ plus(v,r,a) \leftarrow \Delta\theta_1$

$\Downarrow$ dépliage gauche avec la substitution $\{v/b\}$ qui ne change pas $\Delta\theta_1$

$plus(b,r,a) \leftarrow \Delta\theta_1$

Nous cherchons Δ telle que $\Delta\theta_1 = plus(b,r,a)$
$\Delta\{t/0,u/r\} = plus(b,r,a)$ implique $\Delta\{u/r\} = plus(b,r,a)$. Nous obtenons les deux solutions suivantes pour Δ :

$\Delta_1 = plus(b,u,a)$
$\Delta_2 = plus(b,r,a)$

En remplaçant ces valeurs dans C, nous obtenons les formules suivantes :

$plus(t,b,v),\ plus(v,r,a) \leftarrow plus(b,u,a),\ plus(t,r,u)$
$plus(t,b,v),\ plus(v,r,a) \leftarrow plus(b,r,a),\ plus(t,r,u)$

Seule la première formule est valide (elle peut être prouvée par le démonstrateur décrit dans le paragraphe 2). Cette formule exprime une propriété qui combine la commutativité et l'associativité de l'addition dans les entiers naturels.
En remplaçant Δ dans le schéma initial, nous obtenons la définition de div (pour le cas général) :

$div(a,b,s(q),r) \leftarrow plus(b,u,a),\ div(u,b,q,r)$,
Cette définition est correcte si nous supposons que la formule $plus(t,b,v),\ plus(v,r,a) \leftarrow plus(b,u,a),\ plus(t,r,u)$ a été prouvée. C'est la définition que nous avons obtenu dans l'introduction en appliquant la propriété.

4.2 Exemple du tri lent

Soit *SLOW-SORT* le programme suivant spécifiant le tri lent d'une liste d'entiers naturels.

(1)	$sort(x,y)$	$\leftarrow perm(x,y),\ ord(y)$
(2)	$perm(nil,nil)$	$\leftarrow$
(3)	$perm(cons(a,z),cons(b,v))$	$\leftarrow place(b,u,cons(a,z)),\ perm(u,v)$
(4)	$place(b,w,cons(b,w))$	$\leftarrow$
(5)	$place(a_1,cons(b_1,x_1),cons(b_1,y_1))$	$\leftarrow place(a_1,x_1,y_1)$
(6)	$ord(nil)$	$\leftarrow$
(7)	$ord(cons(a,x))$	$\leftarrow lta(a,x),\ ord(x)$
(8)	$lta(x,nil)$	$\leftarrow$
(9)	$lta(x,cons(y,l))$	$\leftarrow lta(x,l),\ inf(x,y)$
(10)	$inf(0,x)$	$\leftarrow$
(11)	$inf(s(x),s(y))$	$\leftarrow inf(x,y)$

Dans cette spécification *nil* et *cons* sont les constructeurs de listes; 0 et s sont les constructeurs des entiers naturels.

$sort(x,y)$ est vrai si et seulement si y est la liste triée de x,
$perm(x,y)$ est vrai si et seulement si y est une permutation de la liste x,
$place(x,y,z)$ est vrai si et seulement si z est la liste obtenue en plaçant l'élément x n'importe où dans la liste y,
$ord(x)$ est vrai si et seulement si la liste x est ordonnée,
$lta(x,l)$ est vrai si et seulement si x est inférieur ou égal à tous les élément de l,
$inf(x,y)$ est vrai si et seulement si $x \leq y$.

1. Génération du schéma.
   Le dépliage de l'atome $perm(x,y)$ de la clause (1) produit les clauses :

   (12) $sort(nil,nil) \leftarrow ord(nil)$
   (13) $sort(cons(a,z),cons(b,v)) \leftarrow place(b,u,cons(a,z)),\ perm(u,v),\ ord(cons(b,v))$

   Seul la clause (13) nous intéresse pour le schéma (la clause (12) fournissant un cas terminal.
   Le pliage partiel de (1) dans (13) génére le schéma suivant :
   $sort(cons(a,z),cons(b,v)) \leftarrow \Delta,\ sort(u,v)$.
   L'objectif est de trouver Δ vérifiant les conditions suivantes:
   - $\mathcal{V}(\Delta) \subset \{a,z,b,u,v\}$.
   - l'ensemble des symboles de prédicats de Δ est un sous-ensemble de $\{place, lta, inf\}$.
2. Instanciation du schéma.
   $\underline{sort(cons(a,z),cons(b,v))} \leftarrow \Delta,\ sort(u,v)$

   $\Downarrow$ dépliage gauche

$perm((cons(a,z),cons(b,v)),\ ord(cons(b,v)) \leftarrow \Delta,\ \underline{sort(u,v)}$

$\Downarrow$ dépliage droit

$perm((cons(a,z),cons(b,v)),\ \underline{ord(cons(b,v))} \leftarrow \Delta,\ perm(u,v),\ ord(v)$

$\Downarrow$ dépliage gauche

$\underline{perm((cons(a,z),cons(b,v))},\ lta(b,v),\ \underline{ord(v)} \leftarrow \Delta,\ perm(u,v),\ \underline{ord(v)}$

$\Downarrow$ dépliage gauche et simplification

$place(b,t,cons(a,z)),\ \underline{perm(t,v)},\ lta(b,v) \leftarrow \Delta,\ \underline{perm(u,v)} \quad (A)$

$\Downarrow$ simplification utilisant la substitution $\{t/u\}$, cette substitution ne change pas Δ car $t \notin \mathcal{V}(\Delta)$.

$place(b,u,cons(a,z)),\ lta(b,v) \leftarrow \Delta$

$\Delta = place(b,u,cons(a,z)),\ lta(b,v)$ est une solution évidente en remplaçant Δ dans le schéma initial et en réarrageant les atomes on trouve la clause suivante :

$$sort(cons(a,z),cons(b,v)) \leftarrow place(b,u,cons(a,z)),\ sort(u,v),\ lta(b,v)$$

Cette définition n'est pas satisfaisante car le test $lta(b,v)$ doit être effectué avant le tri $sort(u,v)$. Remarquons que cette définition peut aussi être obtenue dans le système classique de dépliage-pliage [17].
Ceci nous conduit à chercher une définition récursive de $sort$. La première condition sur les variables de Δ devient $\mathcal{V}(\Delta) \subset \{a,z,b,u\}$.
Nous revenons au point (A)

$place(b,t,cons(a,z)),\ perm(t,v),\ lta(b,v) \leftarrow \Delta,\ \underline{perm(u,v)} \quad (A)$

$\Downarrow$ dépliage droit avec $\theta = \{u/nil,\ v/nil\}$

$$\begin{cases} \pi_1 : place(b,t,cons(a,z)),\ perm(t,nil),\ lta(b,nil) \leftarrow \Delta\theta \\ \pi_2 : (cas\ général) \end{cases}$$

Dans cet exemple, le dépliage droit produit deux formules, la première π_1 par dépliage avec la clause unitaire $perm(nil,nil)$, la seconde π_2 par dépliage avec la clause définissant $perm$ dans le cas général. La suite de la tactique consiste en les deux parties suivantes :

(a) **determiner** une valeur pour Δ en utilisant les cas terminaux.

(b) remplacer Δ par sa valeur dans la formule et la **prouver**

$place(b,t,cons(a,z)),\ \underline{perm(t,nil)},\ lta(b,nil) \leftarrow \Delta\theta$

$\Downarrow$ dépliage gauche, (nous appliquons un dépliage car Δ ne doit pas contenir le prédicat $perm$). Nous utilisons la substitution $\{t/nil\}$

$place(b,nil,cons(a,z)),\ lta(b,nil) \leftarrow \Delta\theta$

puisque $\theta = \{u/nil,\ v/nil\}$ and $v \notin \mathcal{V}(\Delta)$, nous avons

$place(b,nil,cons(a,z)),\ lta(b,nil) \leftarrow \Delta\{u/nil\}$

quatre solutions sont possibles pour Δ :

$\Delta_1 = place(b,u,cons(a,z)),\ lta(b,u)$
$\Delta_2 = place(b,u,cons(a,z)),\ lta(b,nil)$
$\Delta_3 = place(b,nil,cons(a,z)),\ lta(b,u)$
$\Delta_4 = place(b,nil,cons(a,z)),\ lta(b,nil)$

Nous avons généralement à résoudre des équations de la forme $\Gamma \leftarrow \Delta\theta$, où Γ et θ sont donnés et Δ est inconnu. Il peut y avoir plusieurs solutions à de telles équations.
Dans cet exemple, nous choisissons la première solution, celle qui ne contient pas le terme clos nil, $\Delta = \Delta_1$.

En remplaçant Δ par Δ_1 dans (A) nous obtenons :

$place(b,t,cons(a,z)),\ perm(t,v),\ lta(b,v)$
$\leftarrow\ place(b,u,cons(a,z)),\ lta(b,u),\ perm(u,v) \quad (B)$

Cette formule peut être prouvée. En remplaçant Δ dans le schéma initial, nous obtenons la définition suivante de *sort*:

$sort(cons(a,z),cons(b,v)) \leftarrow place(b,u,cons(a,z)),\ lta(b,u),\ sort(u,v)$

qui est un tri par sélection. Il est possible d'obtenir la même définition dans le système classique de dépliage-pliage, à condition d'utiliser la propriété

$$perm(u,v),\ lta(b,v) \leftarrow perm(u,v)\ ,lta(b,u)$$

qui est trés proche de la formule (B), mais le problème est de l'inventer.

5 Discussion et conclusion

Le démonstrateur [16] décrit briévement dans cet article a été implanté en CAML dans le système Spes [2], [6]. On a ainsi pu appliquer la technique avec succès sur plusieurs exemples. Ces expérimentations ont aussi montré d'autres cas ne pouvant être résolus par cette technique. La classe des programmes traités avec succès n'est actuellement pas précisément caractérisée. L'idée générale de la technique que nous avons proposée est la suivante : comme les propriétés sont a priori des formules quelconques, elles sont difficiles à trouver, il faut donc des informations qui permettent de les découvrir. Les informations que nous donnons sont les schémas générés. A l'équation suivante :

dépliages + application de propriétés $\Rightarrow$ pliage (donc définitions récursives)

la technique proposée a substitué l'équation :

dépliages + générations de schémas $\Rightarrow$ génération de propriétés.

La génération de schémas, que nous proposons peut être qualifiée de semi-automatique dans la mesure où un ensemble de schémas est généré et l'on doit en choisir un parmi ceux-ci. Le principal intérêt de cette méthode est de produire des schémas qui sont conformes à la spécification initiale. En effet dans la spécification initiale certains choix d'induction ont été

faits sur les arguments des prédicats, par cette méthode on essaie de trouver des schémas qui "colle" à la spécification. De cette façon on élude partiellement le problème du pliage rencontré dans les approches classiques de dépliage-pliage.

Dans le domaine de la transformation de programmes [12], il n'existe pas beaucoup de travaux traitant de la problématique abordée. Beaucoup des travaux ont été consacrés aux recherches sur l'introduction de nouveaux symboles de prédicats, obtenus en général par des généralisations [13, 14, 1], d'autres travaux basés aussi sur les transformations de pliage-dépliage sont en relation avec l'évaluation partielle [15].

Dans [4, 5] le problème des propriétés est abordé, l'approche est basée sur la notion de programmes bien modés et bien typés. L'idée générale est d'essayer d'obtenir par transformation un programme bien modé. Les variables des programmes obtenus doivent alors vérifier certaines contraintes, ce qui réduit le nombre de formules pouvant jouer le rôle de propriétés. Cette approche peut être complémentaire de celle présentée dans cet article.

Dans le domaine de la synthèse de programmes [7], on peut plus comparer ce travail aux travaux basés sur le concept de "proofs as programs", pour la programmation logique [8]. Cette approche consiste à partir d'une spécification composée d'une formule logique et d'un ensemble d'axiomes sous forme de clauses de Horn, le but est alors de démontrer cette formule en utilisant les axiomes et un démonstrateur se rapprochant de celui que nous avons décrit. Les règles d'inférence permettent de prouver la formule (la spécification) et génèrent simultanément, sous forme de trace, des clauses de Horn qui formeront le programme obtenu par synthèse. Dans cette approche le problème des eurêkas ne se pose pas, ce sont simplement des formules que l'on doit prouver tout au long du processus de preuve. Les inconvénients de cette méthode sont dûs essentiellement au fait qu'il est difficile pour l'utilisateur d'un système basé sur cette méthode, de controler la qualité de la trace obtenue, puisque sa préoccupation première est la preuve. D'autre part l'obtention du programme final nécessite une étape d'extraction à partir de la trace, ce dernier point n'a pas été beaucoup étudié. La problématique de cette approche concerne la preuve. Deux éléments essentiels sont d'une part de trouver les "bons schémas" d'induction, d'autre part de trouver les "bons lemmes". Dans [10], plusieurs sortes d'induction sont comparées. Ce problème du choix du schéma d'induction est un problème délicat dans les approches de construction, synthèse de programmes, ou démonstration par récurrence, il n'existe pas de solution générale satisfaisante à ce problème. Dans [9] on décrit une technique permettant de générer automatiquement des lemmes pour faire des preuves par récurrence. Une étude détaillée prolongeant celle que l'on peut trouver dans [3] permettrait de comparer les lemmes générés dans [9] et dans cet article.

La partie que nous n'avons pas abordée ici est la preuve des formules générées par la tactique. On peut remarquer que ces preuves sont des preuves par récurrence et qu'elles utilisent les règles de coupures, contrairement aux deux premières parties de la tactique. De plus, ce démonstrateur permet aussi de réfuter les formules. Ceci est aussi intéressant car les formules générées ne sont pas toutes valides. L'intérêt de cette technique sera d'autant plus grand que l'on dispose de tactiques et de preuves automatiques, elle pourrait ainsi venir en complément dans un système prenant en compte les nombreuses autres tactiques déjà existantes.

Références

[1] F. Alexandre. A Technique for Transforming Logic Programs by Fold-Unfold Transformations. In M. Bruynooghe and M. Wirsing, editors, *4th International Symposium PLILP'92, Programming Language Implementation and Logic Programming, Leuven Belgium*, volume 631 of *Lecture Notes in Computer Science*, pages 202–216. Springer-Verlag, August 1992.

[2] F. Alexandre, K. Bsaïes, J.P. Finance, and A. Quéré. SPES: A System for Logic Program Transformation. In A. Voronkov, editor, *Proc. of the International Conference on Logic Programming and Automated Reasoning (LPAR'92)*, volume 624 of *Lecture Notes in Artificial Intelligence*, pages 445–447. Springer-Verlag, July 1992. St. Petersburg.

[3] A. Bouverot. *Comparaison entre la transformation et l'extraction de programmes logiques.* PhD thesis, Université Paris VII, 1991.

[4] K. Bsaïes. A Framework for Mechanizing Logic Program Transformation: The Synthesis of Eureka-Properties. In *To appear in the Proc. of ACM SIGPLAN Workshop on Partial Evaluation and semantic-Based Program Manipulation, PEPM 92*, june 19-20 1992. San Francisco, California.

[5] K. Bsaïes. Static Analysis for the Synthesis of Eureka Properties for Transforming Logic Programs. In K. Broda, editor, *4th UK Annual Conference on Logic programming, ALPUK92*, pages 41–61. Workshops in Computing Series, Springer-Verlag, 1992.

[6] K. Bsaïes. *Construction de programmes logiques par synthèse de propriétés.* PhD thesis, Université de Nancy I, Novembre 1993.

[7] Y. Deville and K. K. Lau. Logic Program Synthesis. *Journal of Logic Programming*, 19-20:321–350, 1994.

[8] L. Fribourg. Extracting Logic Programs from Proofs that Use Extended Prolog Execution and Induction. In D.H.D. Warren and P. Szeredi, editors, *7th International Conference on Logic Programming*, pages 685–699, Jerusalem, 1990. MIT Press.

[9] L. Fribourg. Automatic Generation of Simplification Lemmas for Inductive Proofs. In V. Saraswat and K. Ueda, editors, *International Logic Programming Symposium*, pages 103–116. MIT Press, 1991.

[10] L. Fribourg. A Unifying View of Structural Induction and Computation Induction for Logic Programs. In K. K. Lau and T. P. Clement, editors, *LOPSTR'92*, pages 46–60. Workshop in Computing, Springer, 1992.

[11] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[12] A. Pettorossi and M. Proietti. Transformation of Logic Programs: Foundations and techniques. *Journal of Logic Programming*, 19-20:261–320, 1994.

[13] M. Proietti and A Pettorossi. Synthesis of Eureka Predicates for Developing Logic Program. In N. Jones, editor, *3rd European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 306–325, Copenhaguen, 1990. Springer-Verlag.

[14] M. Proietti and A. Pettorossi. The Loop Absorption and Generalization Strategies for the Development of Logic Programs and Partial Deduction. *Journal of Logic Programming*, 16:123–161, 1993.

[15] M. Proietti, A Pettorossi, and S Renault. Reducing nondeterminism while specializing logic programs. In *24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 414–427. ACM Press, January 1997.

[16] A. Sakurai and H. Motoda. Proving Definite Clauses without Explicit Use of Inductions. In K. Furukawa, H. Tanaka, and T Fujisaki, editors, *Proceedings of the 7th Conference, Logic Programming '88*, volume 383 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1988.

[17] H. Tamaki and T. Sato. Unfold/Fold Transformation of Logic Programs. In *Proceedings of the 2nd International Logic Programming Conference*, Uppsala, 1984.